Parallel Rcpp

Cecina Babich Morrow

2024-02-09

Parallel local regression

In this exercise, we will use a dataset from an Irish smart meter:

```r
# If necessary, install electBook
if(!require(electBook)) {
  # Install electBook from GitHub
  library(devtools)
  install_github("mfasiolo/electBook")
}

# Load packages
library(electBook)
library(tidyverse)

data(Irish)
```

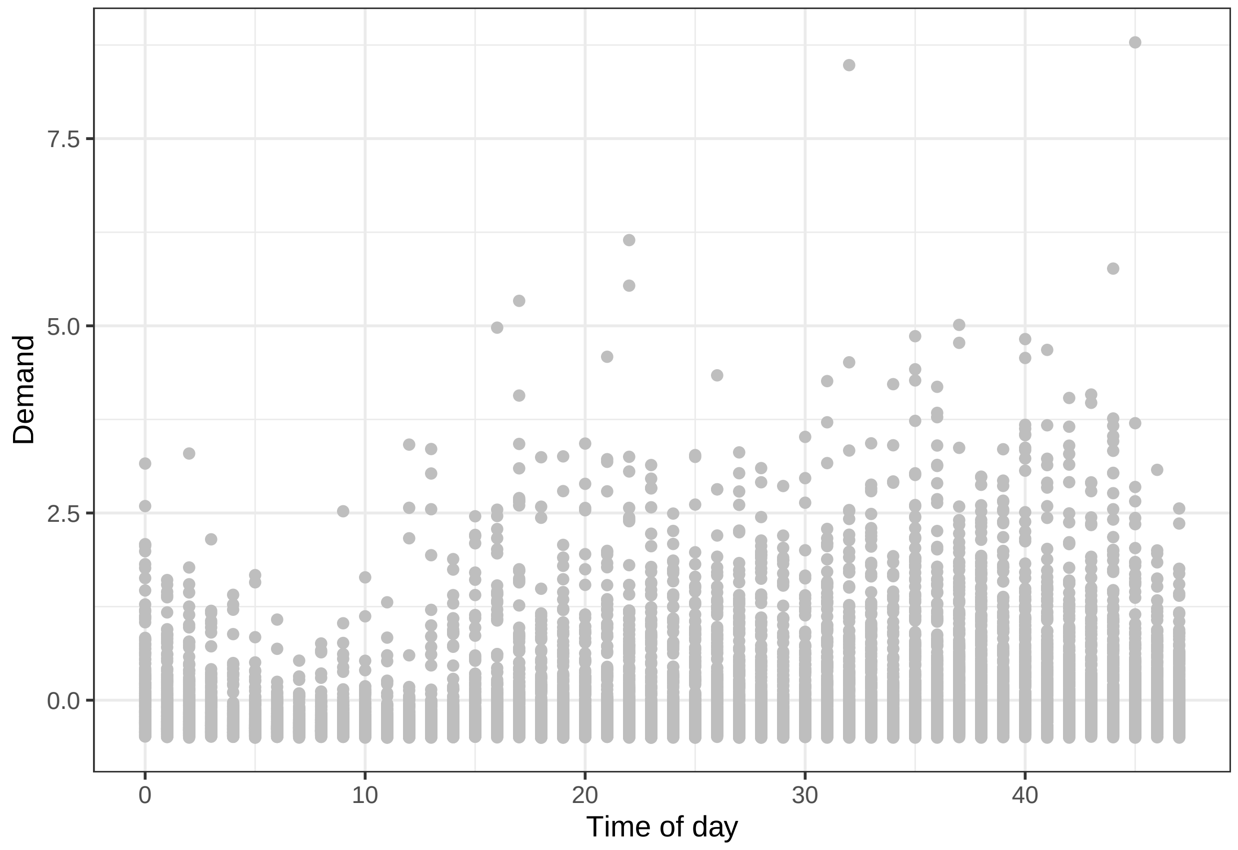
We will concatenate the electricity demand from all households into the vector y:

```r
y <- do.call("c", Irish$indCons)
y <- y - mean(y)
str(y)
```

```
##  Named num [1:44886928] -0.477 -0.366 -0.405 -0.476 -0.366 ...
##  - attr(*, "names")= chr [1:44886928] "I10021" "I10022" "I10023" "I10024" ...
```

We can plot a subset of the demand over time of day (on the x-axis):

```r
ncust <- ncol(Irish$indCons)

x <- rep(Irish$extra$tod, ncust)

n <- length(x)
ss <- sample(1:n, 1e4)
subset_data <- data.frame(x = x[ss], y = y[ss])
ggplot(subset_data, aes(x = x, y = y)) +
  geom_point(color = "grey") +
  labs(x = "Time of day", y = "Demand") +
  theme_bw()
```



Linear regression

We want to use the time of day x to predict demand y. We will start with the model $\mathbb{E}(y|x) = \beta_1 x$. This function fits the model using least squares:

```r
reg1D <- function(y, x){
  b <- t(x) %*% y / (t(x) %*% x)
  return(b)
}
```

We can compare the speed of our function with `lm`:

```r
# Compare speed
# lm
system.time( lm(y ~ -1 + x)$coeff )[3]
```

```
## elapsed
##  21.841
```

```r
# reg1D
system.time( reg1D(y, x) )[3]
```

```
## elapsed
##   1.146
```

```r
# Calculate beta_1 using reg1D
beta1 <- reg1D(y, x)
# Check if the result from reg1D is equal to the result from lm
lm_beta1 <- lm(y ~ -1 + x)$coeff
```

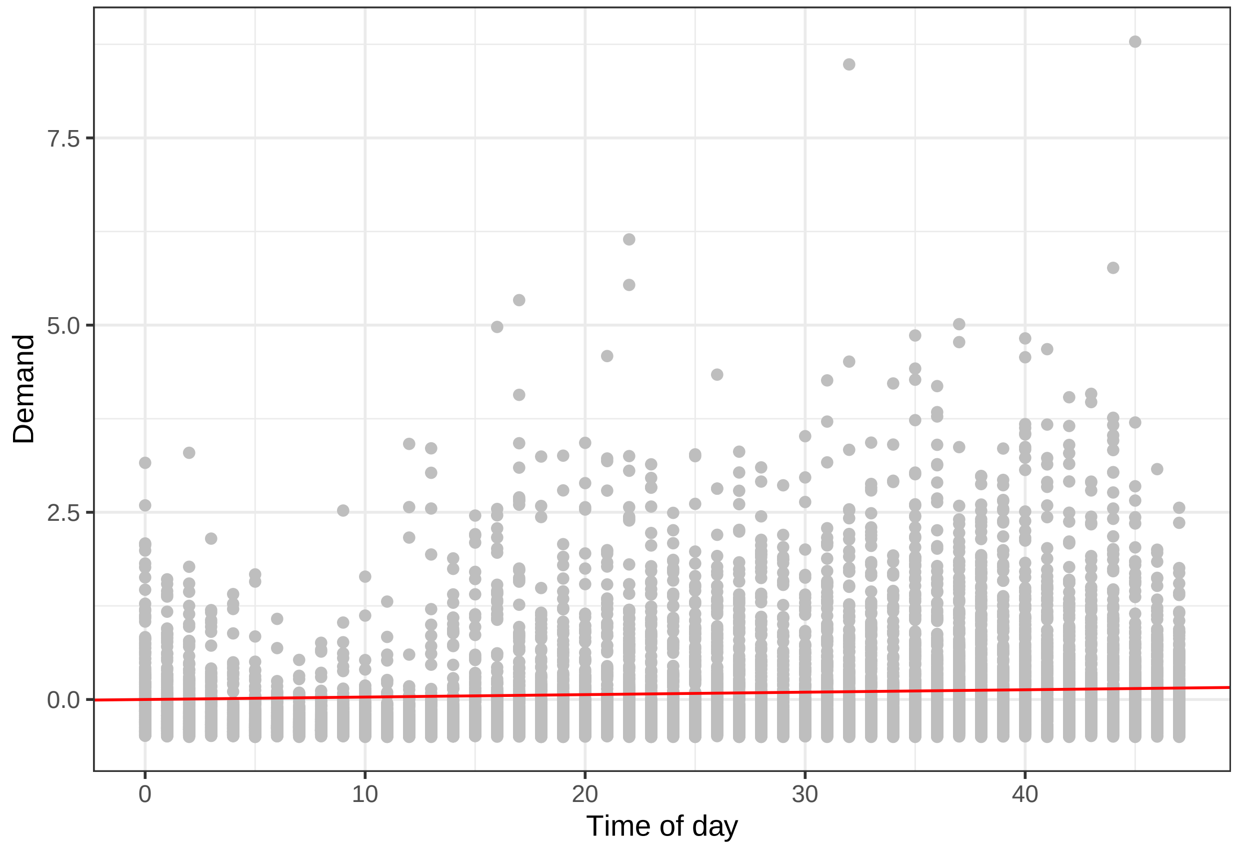
```r
all.equal(beta1[1,1], as.numeric(lm_beta1))
```

```
## [1] TRUE
```

We see that our function is much faster than `lm` (probably because we aren't computing as many values). Our result for β_1 is the same as the result from `lm`.

We can visualize the resulting fit:

```r
ggplot(subset_data, aes(x = x, y = y)) +
  geom_point(color = "grey") +
  geom_abline(slope = beta1, intercept = 0, color = "red") +
  labs(x = "Time of day", y = "Demand") +
  theme_bw()
```



This is not a particularly good fit.

Question 1

We now want to implement a parallel version of `reg1D` using `RcppParallel` to see if we can achieve even faster performance.

```cpp
#include <Rcpp.h>
#include <omp.h>
using namespace Rcpp;

// [[Rcpp::plugins(openmp)]]
// [[Rcpp::export]]
double reg1D_parallel(NumericVector y, NumericVector x) {
  int n = y.size();
  double xTy = 0.0;
  double xTx = 0.0;
  #pragma omp parallel for reduction(+:xTy, xTx)
  for(int i = 0; i < n; i++) {
    xTy += x[i] * y[i];
    xTx += x[i] * x[i];
  }

  return xTy / xTx;
}
```

We can check to make sure our new parallelized function yields the same results as `reg1D`, and then compare the performance of the two functions:

```
beta1_parallel <- reg1D_parallel(y, x)
all.equal(beta1_parallel, beta1[1,1])
```

```
## [1] TRUE
```

```
# Compare speed
system.time( reg1D(y, x) )[3]
```

```
## elapsed
##   1.423
```

```
system.time( reg1D_parallel(y, x) )[3]
```

```
## elapsed
##   0.041
```

We see that our parallelized function is faster than the original function, and it yields the same (poor) results.

Polynomial regression

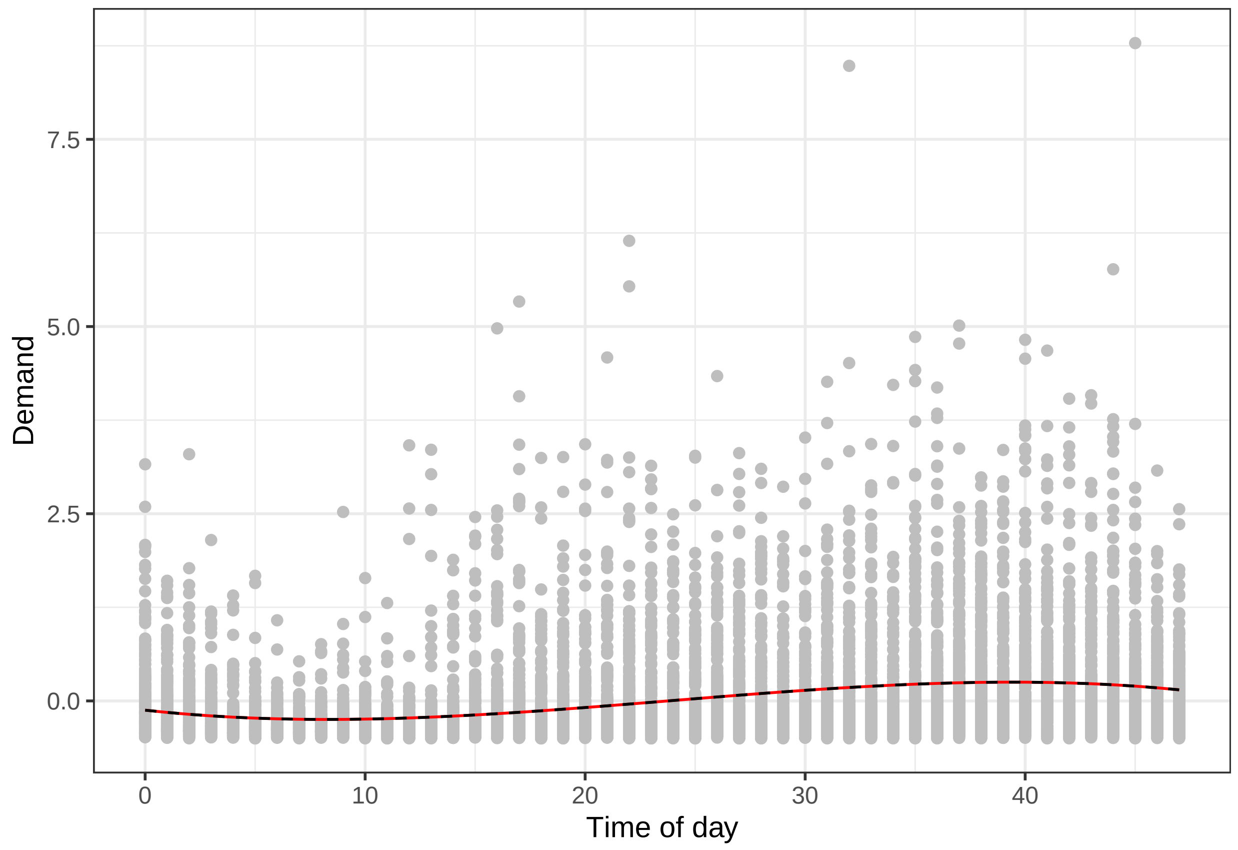
We want to improve upon the fit of our linear regression by fitting the polynomial regression $\mathbb{E}(y|x) = \beta_0 + \beta_1 x + \beta_2 x^2 + \beta_3 x^3$. We can write a function that calculates the Cholesky decomposition $\mathbf{C}^\top \mathbf{C} = \mathbf{X}^\top \mathbf{X}$ where $\mathbf{C}$ is upper triangular. Then $\hat{\beta} = \mathbf{C}^{-1}\mathbf{C}^{-\top}\mathbf{X}^\top \mathbf{y}$. This can be solved by computing $\mathbf{z} = \mathbf{C}^{-\top}\mathbf{X}^\top \mathbf{y}$ by forward-solving the lower-triangular system and then $\hat{\beta} = \mathbf{C}^{-1}\mathbf{z}$ by back-solving the upper-triangular system:

```
regMD <- function(y, X){
  XtX <- t(X) %*% X
  C <- chol(XtX)
  z <- forwardsolve(t(C), t(X) %*% y)
  b <- backsolve(C, z)
  return(b)
}
```

We can test out our function:

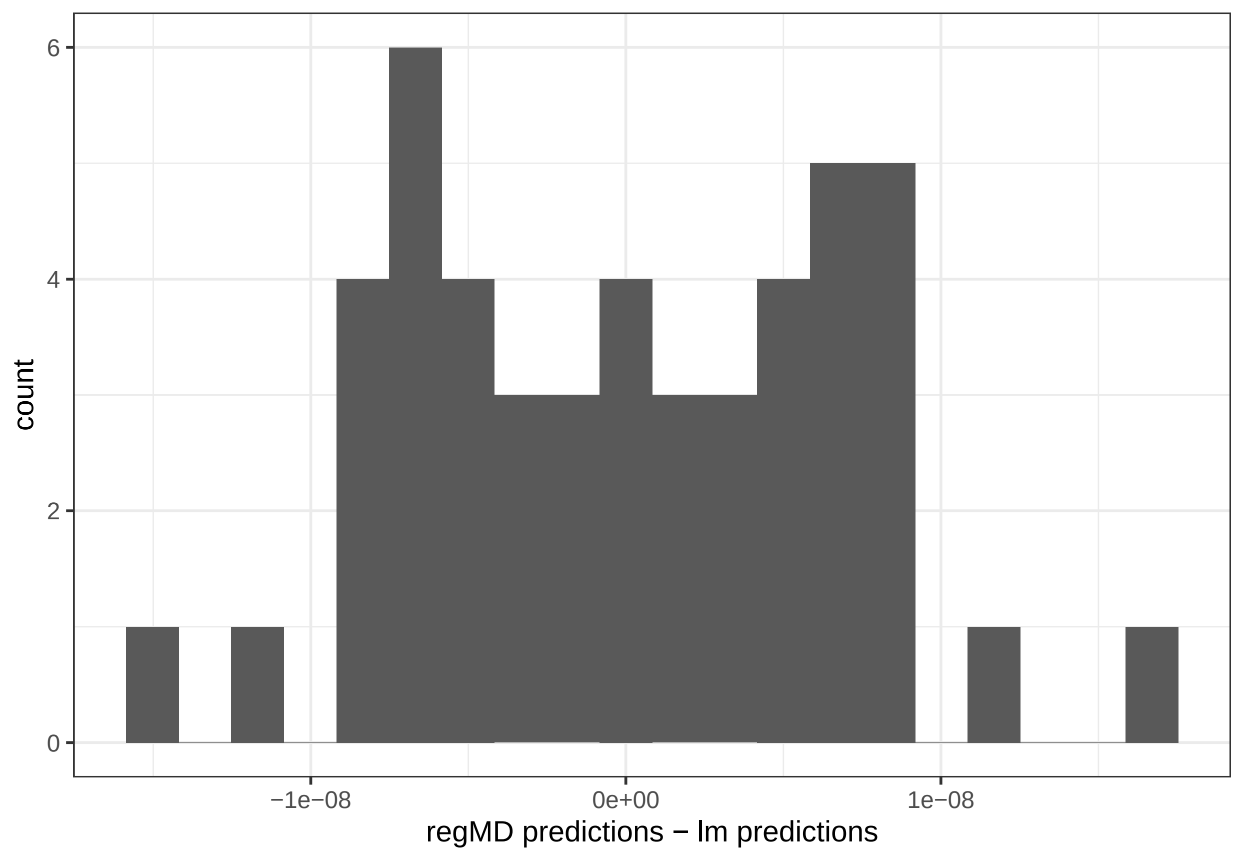
```
X <- cbind(1, x, x^2, x^3)
Beta <- regMD(y, X)

# Compare to beta from lm
Beta_lm <- lm(y ~ -1 + X)$coeff

Xu <- cbind(1, 0:47, (0:47)^2, (0:47)^3)
```

```r
poly_preds <- data.frame(x = 0:47,
                         regMD_preds = Xu %*% Beta,
                         lm_preds = Xu %*% Beta_lm) %>%
  mutate(regMD_minus_lm = regMD_preds - lm_preds)

# Compare values from regMD and lm
ggplot(subset_data, aes(x = x, y = y)) +
  geom_point(color = "grey") +
  geom_line(data = poly_preds, aes(x = x, y = regMD_preds),
            color = "red") +
  geom_line(data = poly_preds, aes(x = x, y = lm_preds),
            color = "black", linetype = "dashed") +
  labs(x = "Time of day", y = "Demand") +
  theme_bw()
```



We can see that the fits are almost identical (and much better than the linear regression). We can compare the exact values to see that we have slightly different predictions:

```r
# Compare values from regMD and lm
ggplot(data = poly_preds, aes(x = regMD_minus_lm)) +
  geom_histogram(bins = 20) +
  labs(x = "regMD predictions - lm predictions") +
  theme_bw()
```



Question 2

We now want to implement a parallel version of `regMD` using `RcppParallel`.

```cpp
// [[Rcpp::depends(RcppArmadillo)]]
#include <RcppArmadillo.h>
#include <omp.h>
using namespace Rcpp;
using namespace arma;

// [[Rcpp::plugins(openmp)]]
// [[Rcpp::export]]
NumericVector regMD_parallel(NumericVector y, NumericMatrix X) {
  int n = y.size();
  int p = X.ncol();
  NumericMatrix XtX(p, p);
  NumericVector Xty(p);

  // Compute XtX in parallel
  #pragma omp parallel for collapse(2)
  for(int i = 0; i < p; i++) {
    for(int j = 0; j < p; j++) {
      for(int k = 0; k < n; k++) {
        XtX(i, j) += X(k, i) * X(k, j);
      }
    }
  }
  // Convert XtX to an armadillo matrix
  arma::mat XtX_arma = as<arma::mat>(XtX);

  // Compute C
  arma::mat C_arma = chol(XtX_arma, "upper");

  // Compute Xty in parallel
  #pragma omp parallel for
  for(int i = 0; i < p; i++) {
    for(int k = 0; k < n; k++) {
      Xty[i] += X(k, i) * y[k];
    }
  }

  // Compute z using forward substitution
  arma::vec Xty_arma = as<arma::vec>(Xty);
  arma::vec z = solve(C_arma.t(), Xty_arma);

  // Compute b using back substitution
  arma::vec b = solve(C_arma, z);
  // Convert b to NumericVector
  NumericVector b_Rcpp = wrap(b);

  return b_Rcpp;
}
```

We check to make sure that `regMD_parallel` yields the same results as `regMD`, and then compare the performance of `lm`, `regMD`, and `regMD_parallel`:

```
Beta_parallel <- regMD_parallel(y, X)
all.equal(Beta_parallel, Beta)
```

```
## [1] TRUE
```

```
# Compare speeds
system.time( lm(y ~ -1 + X)$coeff )[3]
```

```
## elapsed
##  23.246
```

```
system.time( regMD(y, X) )[3]
```

```
## elapsed
##   6.672
```

```
system.time( regMD_parallel(y, X) )[3]
```

```
## elapsed
##   0.918
```

We once again see that our initial function is much faster than using `lm`, and our parallelized function is the fastest of all.

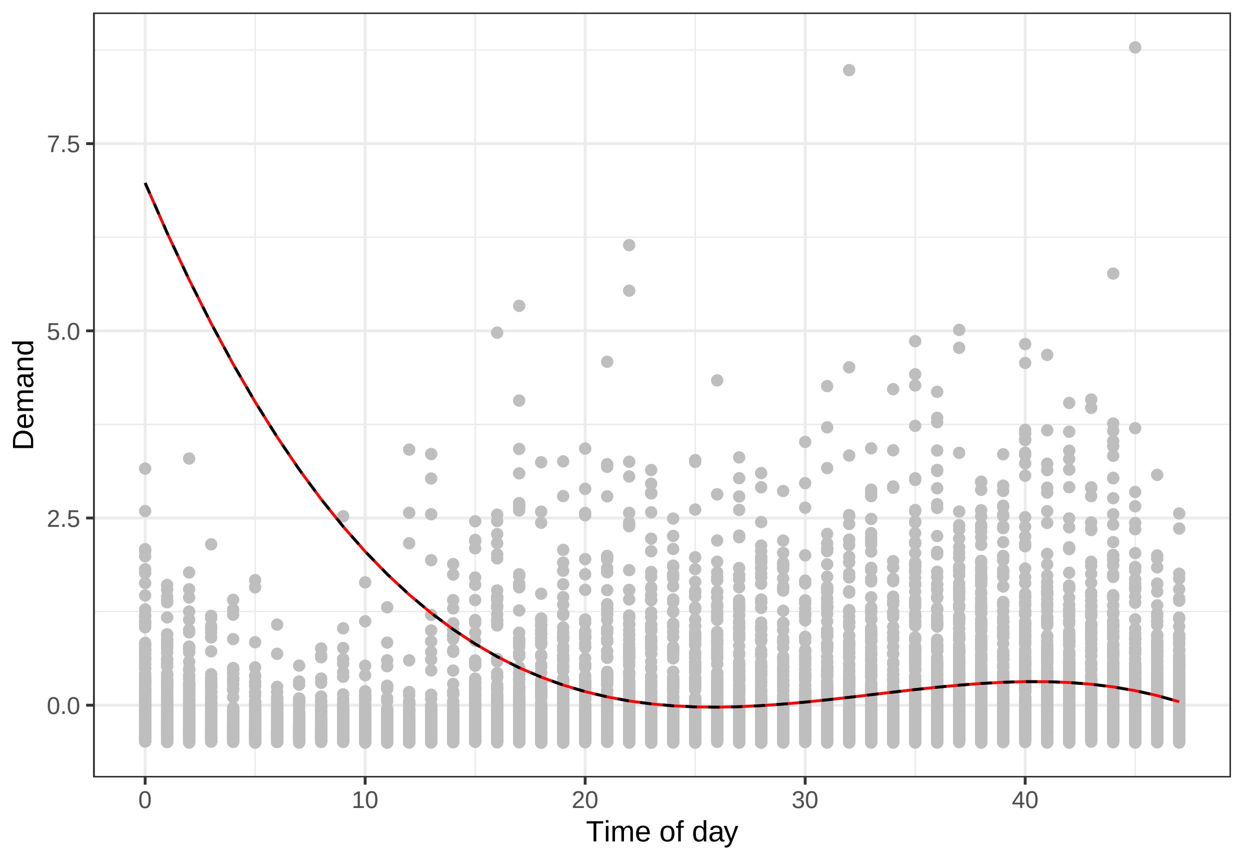
Local polynomial regression

The following function fits a basic one-dimensional local polynomial regression using a Gaussian kernel:

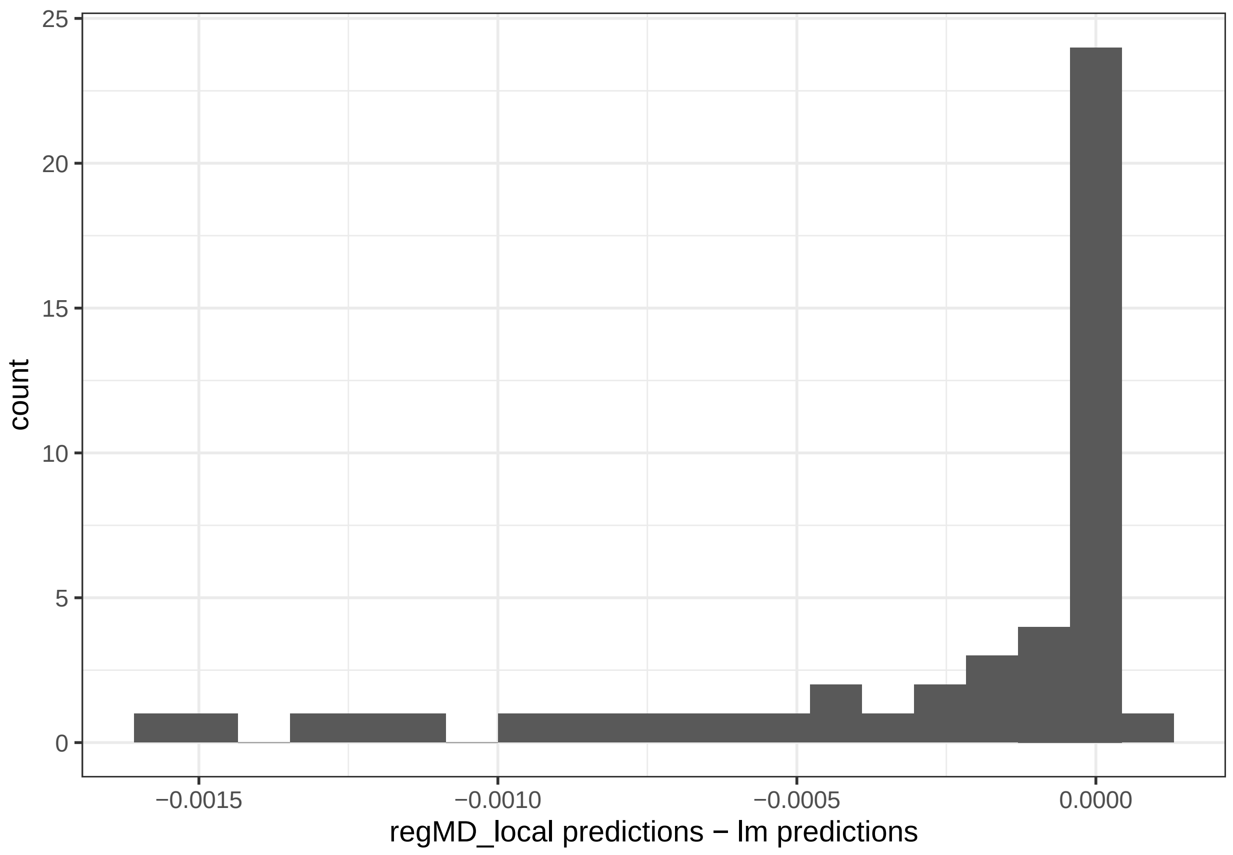
```r
regMD_local <- function(y, X, x0, x, h){
  # Weights
  w <- dnorm(x0, x, h)
  w <- w / sum(x) * length(y) # Normalize weights

  wY <- y * sqrt(w)
  wX <- X * sqrt(w)
  wXtwX <- t(wX) %*% wX

  C <- chol(wXtwX)

  z <- forwardsolve(t(C), t(wX) %*% wY)

  b <- backsolve(C, z)

  return(b)
}
```

We can check our results against `lm` again:

```r
b_lpr <- regMD_local(y, X, x0 = 35, x = x, h = 5)
lm_b_lpr <- lm(y ~ -1 + X, weights = dnorm(35, x, 5))$coeff

lpr_preds <- data.frame(x = 0:47,
                        regMD_local_preds = Xu %*% b_lpr,
                        lm_preds = Xu %*% lm_b_lpr) %>%
  mutate(regMD_local_minus_lm = regMD_local_preds - lm_preds)

# Compare values from regMD_local and lm
ggplot(subset_data, aes(x = x, y = y)) +
  geom_point(color = "grey") +
  geom_line(data = lpr_preds, aes(x = x, y = regMD_local_preds),
            color = "red") +
  geom_line(data = lpr_preds, aes(x = x, y = lm_preds),
            color = "black", linetype = "dashed") +
  labs(x = "Time of day", y = "Demand") +
  theme_bw()
```



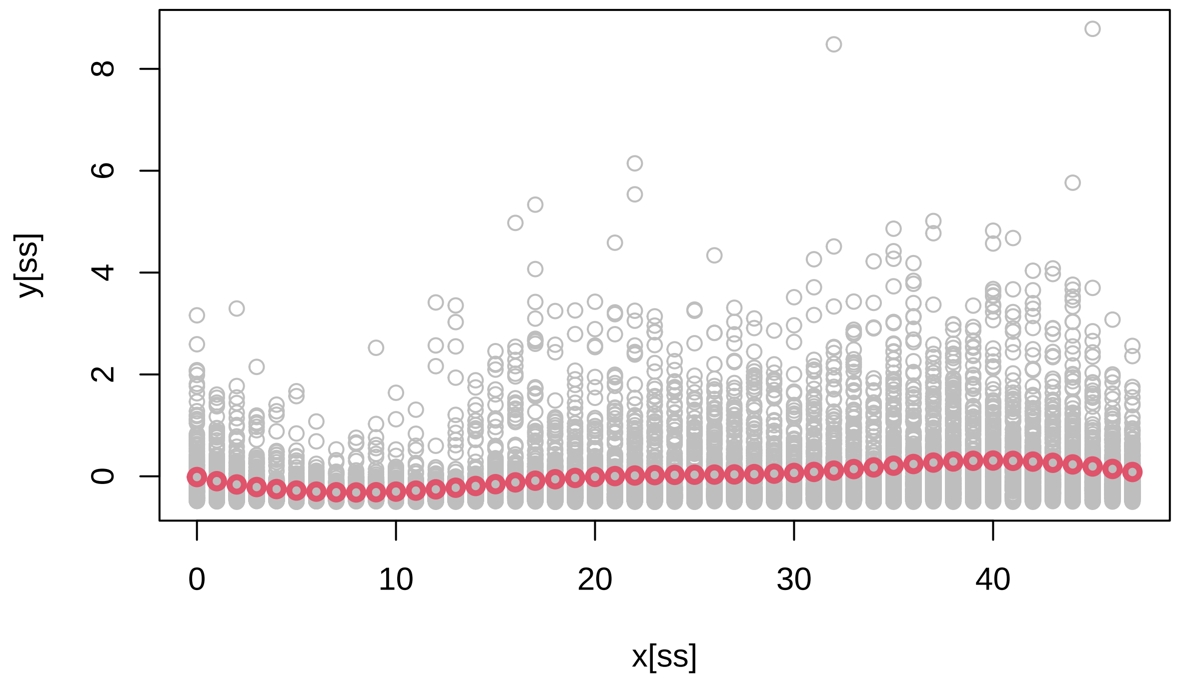
```
ggplot(data = lpr_preds, aes(x = regMD_local_minus_lm)) +
  geom_histogram(bins = 20) +
  labs(x = "regMD_local predictions - lm predictions") +
  theme_bw()
```



We see very similar results to `lm`, but it does seem like our `regMD_local` function skews slightly lower than the predictions from `lm`.

The following code evaluates the fitted function at various times of day (but it does take awhile to run):

```r
f <- lapply(0:47, function(.x ) regMD_local(y, X, x0 = .x, x = x, h = 5))

plot(x[ss], y[ss], col = "grey")
lines(0:47, sapply(1:48, function(ii) Xu[ii, ] %*% f[[ii]]), col = 2, lwd = 3, type = 'b')
```



We see a pretty reasonable prediction.

Question 3

We now want to implement a parallel version of `regMD_local` using `RcppParallel`.

```
// [[Rcpp::depends(RcppArmadillo)]]
#include <RcppArmadillo.h>
#include <omp.h>
using namespace Rcpp;
using namespace arma;

// [[Rcpp::plugins(openmp)]]
// [[Rcpp::export]]
vec regMD_local_parallel(NumericVector y,
                         NumericMatrix X,
                         double x0,
                         NumericVector x,
                         double h) {
  int n = y.size();
  int p = X.ncol();
  NumericVector w(n);
  NumericMatrix wX(n, p);
  NumericVector wY(n);

  // Compute weights in parallel
  // Compute sum of x
```

```
double sum_x = sum(x);
#pragma omp parallel for
for(int i = 0; i < n; i++) {
  w[i] = R::dnorm(x0, x[i], h, 0) / sum_x * n;
}

// Compute wX and wY in parallel
#pragma omp parallel for
for(int i = 0; i < n; i++) {
  double sqrt_w = sqrt(w[i]);
  for(int j = 0; j < p; j++) {
    wX(i, j) = X(i, j) * sqrt_w;
  }
  wY[i] = y[i] * sqrt_w;
}

// Compute wXtwX in parallel
NumericMatrix wXtwX(p, p);
#pragma omp parallel for collapse(2)
for(int i = 0; i < p; i++) {
  for(int j = 0; j < p; j++) {
    for(int k = 0; k < n; k++) {
      wXtwX(i, j) += wX(k, i) * wX(k, j);
    }
  }
}

// Convert wXtwX to an armadillo matrix
arma::mat wXtwX_arma = as<arma::mat>(wXtwX);

// Compute C
arma::mat C_arma = chol(wXtwX_arma, "upper");

// Compute wXtwy in parallel
NumericVector wXtwy(p);
#pragma omp parallel for
for(int i = 0; i < p; i++) {
  for(int k = 0; k < n; k++) {
    wXtwy[i] += wX(k, i) * wY[k];
  }
}

// Compute z using forward substitution
arma::vec wXtwy_arma = as<arma::vec>(wXtwy);
arma::vec z = solve(C_arma.t(), wXtwy_arma);

// Compute b using back substitution
arma::vec b = solve(C_arma, z);
// Convert b to NumericVector
NumericVector b_Rcpp = wrap(b);

return b_Rcpp;
```

```
}
```

We check to make sure that `regMD_local_parallel` yields the same results as `regMD_local`, and then compare the performance:

```
b_lpr_parallel <- regMD_local_parallel(y, X, x0 = 35, x = x, h = 5)
all.equal(b_lpr_parallel, b_lpr)
```

```
## [1] TRUE
```

```
# Compare speeds
system.time( lm(y ~ -1 + X, weights = dnorm(35, x, 5))$coeff )[3] # 13.142
```

```
## elapsed
##  15.906
```

```
system.time( regMD_local(y, X, x0 = 35, x = x, h = 5) )[3] # 3.218
```

```
## elapsed
##    3.27
```

```
system.time( regMD_local_parallel(y, X, x0 = 35, x = x, h = 5) )[3] # 2.288
```

```
## elapsed
##   1.136
```

Weight normalization

Consider the following:

```
x_test <- c(100, 200, 300)
w <- dnorm(0, x_test, 1)
(w <- w / sum(x_test))
```

```
## [1] 0 0 0
```

All of the weights are 0 because the values of `x_test` are very far from 0 with a standard deviation of only 1. This would prevent us from being able to use local regression since our weight vector would be all 0s. Ideally, we would like all the weights to sum to 1.

To compute this in a numerically stable way, we do the following:

```
lw <- dnorm(0, x_test, 1, log = TRUE)
mlw <- max(lw)
( w <- exp(lw - mlw) / sum(exp(lw - mlw)) )
```

```
## [1] 1 0 0
```

By applying a log transformation, we can manage the very small weights resulting from `dnorm` for these values. This solution is preferable because it is numerically stable.

Question 4

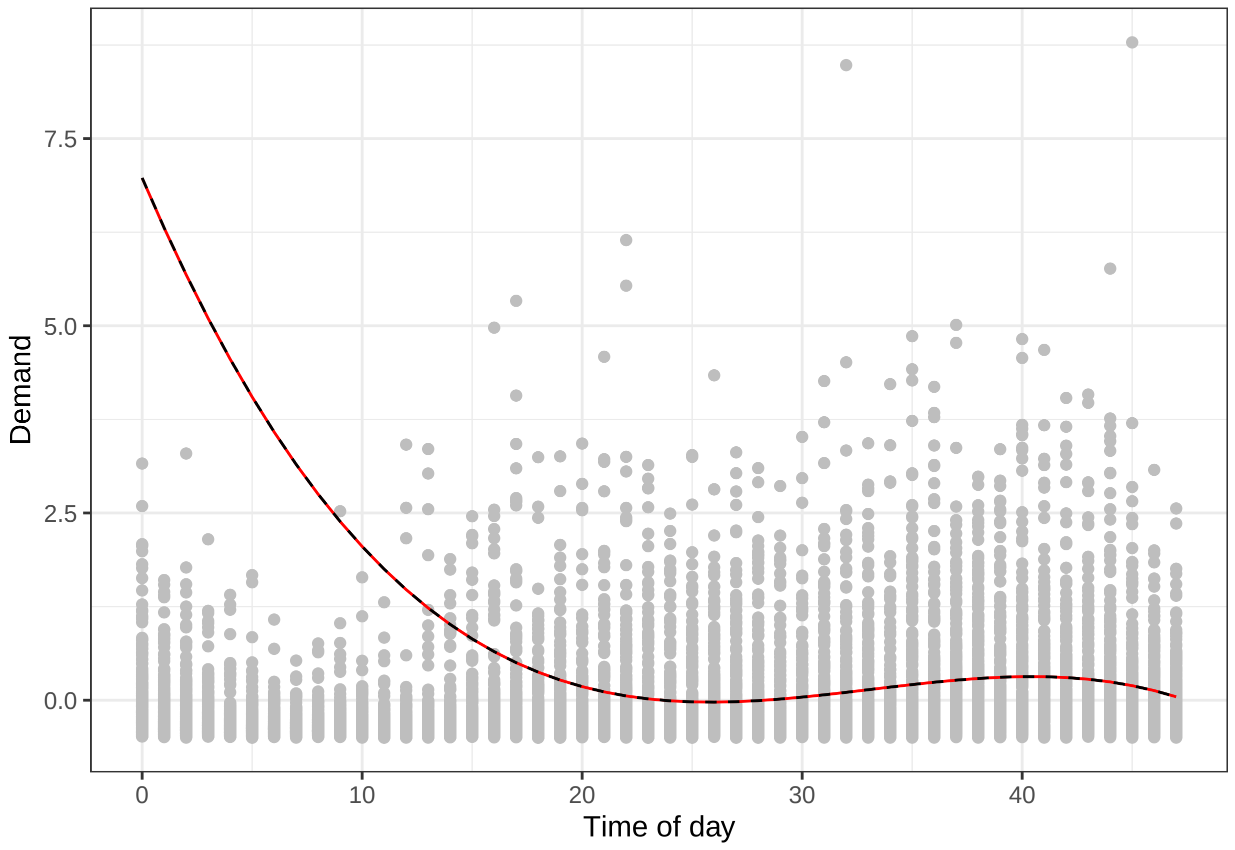
We now want to implement a parallel version of this.

```
// [[Rcpp::depends(RcppArmadillo)]]
#include <RcppArmadillo.h>
#include <omp.h>
using namespace Rcpp;
using namespace arma;
```

```
// [[Rcpp::plugins(openmp)]]
// [[Rcpp::export]]
arma::vec regMD_local_stable_parallel(NumericVector y,
                                      NumericMatrix X,
                                      double x0,
                                      NumericVector x,
                                      double h) {
  int n = y.size();
  int p = X.ncol();
  arma::vec w(n);
  arma::mat wX(n, p);
  arma::vec wY(n);

  // Compute log weights in parallel
  arma::vec lw(n);
  #pragma omp parallel for
  for(int i = 0; i < n; i++) {
    lw[i] = R::dnorm(x0, x[i], h, 1);
  }
  double mlw = max(lw);

  // Compute weights in parallel
  arma::vec exp_lw(n);
  double sum_exp_lw = 0;
  #pragma omp parallel for reduction(+:sum_exp_lw)
  for(int i = 0; i < n; i++) {
    exp_lw[i] = exp(lw[i] - mlw);
    sum_exp_lw += exp_lw[i];
  }
  w = exp_lw / sum_exp_lw;

  // Compute wX and wY in parallel
  #pragma omp parallel for
  for(int i = 0; i < n; i++) {
    double sqrt_w = sqrt(w[i]);
    for(int j = 0; j < p; j++) {
      wX(i, j) = X(i, j) * sqrt_w;
    }
    wY[i] = y[i] * sqrt_w;
  }

  // Compute wXtwX in parallel
  NumericMatrix wXtwX(p, p);
  #pragma omp parallel for collapse(2)
  for(int i = 0; i < p; i++) {
    for(int j = 0; j < p; j++) {
      for(int k = 0; k < n; k++) {
        wXtwX(i, j) += wX(k, i) * wX(k, j);
      }
    }
  }

  // Compute wXtwy in parallel
  NumericVector wXtwy(p);
  #pragma omp parallel for
  for(int i = 0; i < p; i++) {
    for(int k = 0; k < n; k++) {
      wXtwy[i] += wX(k, i) * wY[k];
    }
  }

  // Convert wXtwX and wXtwy to armadillo
  arma::mat wXtwX_arma = as<arma::mat>(wXtwX);
  arma::vec wXtwy_arma = as<arma::vec>(wXtwy);

  // Compute C
  arma::mat C = chol(wXtwX_arma, "upper");
  // Compute z using forward substitution
  arma::vec z = solve(C.t(), wXtwy_arma);
  // Compute b using back substitution
  arma::vec b = solve(C, z);

  return b;
}
```

We check to see if `regMD_local_stable_parallel` gives similar predictions:

```
b_lpr_stable_parallel <- regMD_local_stable_parallel(y, X, x0 = 35, x = x, h = 5)

lpr_preds <- lpr_preds %>%
  mutate(regMD_local_stable_preds = Xu %*% b_lpr_stable_parallel)

# Compare values from regMD_local_stable_parallel and regMD_local
ggplot(subset_data, aes(x = x, y = y)) +
  geom_point(color = "grey") +
  geom_line(data = lpr_preds, aes(x = x, y = regMD_local_preds),
            color = "red") +
  geom_line(data = lpr_preds, aes(x = x, y = regMD_local_stable_preds),
            color = "black", linetype = "dashed") +
  labs(x = "Time of day", y = "Demand") +
  theme_bw()
```



The red line shows the prediction from `regMD_local` and the black dashed line shows the prediction from `regMD_local_stable_parallel`. We see that the predictions are almost identical.